Putting Non-Functional Requirements into Software Architecture[1]

Xavier Franch, Pere Botella
{franch, botella}@lsi.upc.es
Dept. Llenguatges i Sistemes Informàtics (LSI)
Universitat Politècnica de Catalunya (UPC)
c/Jordi Girona 1-3 (Campus Nord), 08034 Barcelona, Catalonia (Spain)
FAX: 34-3-4017014. Phone: 34-3-4016965 / 6960

Abstract

This paper presents an approach for incorporating non-functional information of software systems into software architectures. To do so, components present two distinguished slots: their non-functional specification, where non-functional requirements on components are placed, and their non-functional behaviour with respect to these requirements. Also, connector protocols may describe which non-functional aspects are relevant to component connections. We propose a notation to describe non-functionality in a systematic manner, and we use it to analyse two particular aspects of the meeting scheduler case study, user interaction and performance.

1. Introduction

Software systems are characterised both by their *functionality* (what the system does) and by their *non-functionality* or quality (how the system behaves with respect to some observable attributes like performance, reusability, reliability, etc.). Both aspects are relevant to software development but, traditionally, much more work has be done for the first one than for the second, especially concerning specification languages and, lately, also architectural languages. This happens in despite of the fact that many researchers have pointed out the convenience of non-functional features appearing in those languages [5, 15, 18, 19, 22].

In this paper, we are going to propose a way of putting non-functional information of software systems into software architectures. So long as we are not interested in a particular software architectural style or notation, we will use an ad-hoc language for describing components and connectors, and we will focus on the non-functional part, by explaining how we incorporate it into components and connectors and by presenting a language called NoFun to describe it. We will illustrate the feasibility of our approach by addressing two particular aspects of the meeting scheduler case study.

Before introducing our proposal, we need to clarify our point of view about non-functionality. We distinguish three key concepts:

- *Non-functional attribute* (short, *NF-attribute*): any attribute of software which serves as a mean to describe it and possibly to evaluate it. Among the most widely accepted [3, 4, 13, 14, 17] we can mention: time and space efficiency, reusability, maintainability, reliability and usability. In our approach, we allow arbitrary identification and definition of NF-attributes.
- *Non-functional behaviour* (short, *NF-behaviour*): any assignment of values to the NF-attributes that are in use in a particular software unit (component, connector, port, etc.).
- *Non-functional requirement* (short, *NF-requirement*): any constraint referred to a subset of the NF-attributes that are in use in a particular software unit.

[1] This work is partially supported by the spanish project TIC97-1158 (from the CICYT program).

2. A Model for Software Architecture

Our model for software architecture distinguishes two usual kinds of units, components and connectors.

2.1 Components

According to [1], "components are the active, computational entities of a system... The relationship between a component and its environment is defined explicitly as a collection of interaction points, or *ports*". In our current framework, we restrict ourselves to software components, although we feel that most of the ideas presented here could be applied also to physical components without significant differences.

Basic software components are software modules (classes), but others can be defined: libraries of reusable components, whole systems, clusters of similar systems, etc. However, we do not fix the set of valid software components.

In our software architecture style, components have a specification and an implementation, each one with a functional part and a non-functional one.

- The functional specification describes the services that the component provides. To be accurate, functional specifications should be stated using any of the existing specification languages, like Z [20] and Larch [9] for sequential systems, or CSP [12] and CHAM [2] for more general ones.
- The non-functional specification declares which NF-attributes are of interest for the component and establishes properties concerning them by means of NF-requirements. In fact, NF-attributes will be often defined in separate and independent modules, called *attribute modules*, to be reused in many contexts.
- The functional implementation is the operational artefact that makes the functional specification to work out. It may be compound, to obtain the implementation of a component in terms of a software architecture that glues simpler ones. Usual programming languages can be used for implementation purposes.
- The non-functional implementation is a description of the NF-behaviour of the functional implementation with respect to the NF-attributes declared in the non-functional specification.

In fact, much of a component specification come from its classification in a component type, which provide some implicit properties; it is sometimes said that a component is an instance of its component type. For instance, a filter is a type of component that carries out a manipulation of input data and renders the result to an output port. Some useful taxonomies of components exist in the literature (e.g., [10, 11]). Although we plan to do it in the future, in the current state of our work we have not bound yet non-functional information to any of these component types, but directly to component instances.

Non-functional issues will be introduced in architectures using the language presented in section 3.

2.2. Connectors

Once again, we adopt the definition appearing in [1] that states that "connectors define the interaction between components. Each connector provides a way for a collection of ports to come into contact and logically defines the protocol through which a set of components will interact". Connectors can be then as diverse as: method invocation, network connection, data sharing, etc.

In any case, and not surprisingly, connectors can be also divided into the same parts as components. Functional specification can define things as: network connection protocol, parameter passing mechanism, etc. Non-functional specification will introduce NF-attributes as: reliability of connection, efficiency, etc., together with NF-requirements. Implementations will work as in components.

Fig. 1 shows an example of software architecture (borrowed from [1]) which is an instance of the pipe-filter architectural style: components are filters, while connectors are pipes supporting dataflow between two filters. Pipes connects filters using the appropriate ports; other ports remain free, standing for external connections of the resulting system.

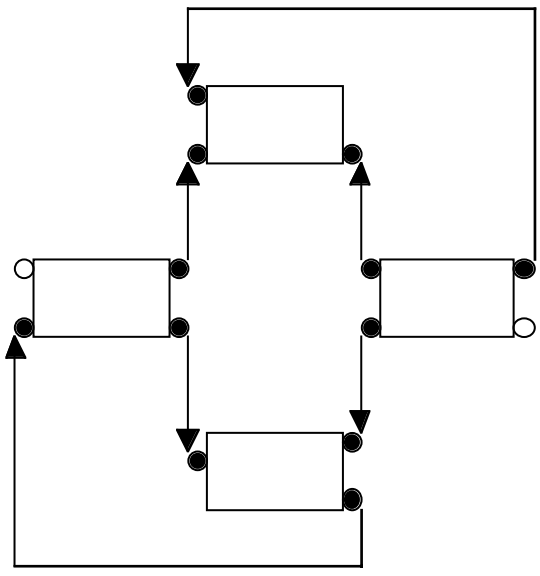


Fig. 1: A pipe-filter software architecture.

3. The Language NoFun

In this section, we present the highlights of a language called *NoFun* aimed at stating non-functional information into software components and connectors. In fact, NoFun constructs will be really introduced in the next section when dealing with the case study, although just to the extend they are needed. A more detailed definition can be found at [8].

3.1. Non-functional attributes

In addition to their name, NF-attributes have the following characteristics in NoFun:

- They belong to a *domain*, which fixes the set of valid values and operations. Domains are then similar to the usual notion of data type in a programming language and so we include integer and real numbers (for measurable NF-attributes, for instance degree of reusability), boolean values (for NF-attributes that just hold or fail, as error recovery), disjoint union (for instance, efficiency as the disjoint union of time and space efficiency), by enumeration of values and so on. Also, we consider the existence of ad-hoc domains for particular cases of software systems; for instance, we have defined a domain for asymptotic complexity for measuring efficiency of data structures.
- Their are classified of one of two *kinds*, basic or derived, depending on whether their value can be computed from others or not. A derived NF-attribute P includes the following parts:
  - ◊ The list L of other NF-attributes that determine P's value.
  - ◊ A list of n guarded formulae of the form $C_i => P = E_i$, $1 \le i \le n$, C_i being a boolean expression and E_i an expression yielding a value in P's domain; if $n = 1$, then C_i is optional. The meaning of a formula is: P equals E_i if the condition C_i holds.
- They can be *bound* to different kind of components (program modules, libraries, systems, etc.), or even to particular services provided by components by means of ports.
- They have a *scope*, which determines the components in which they are in use. There is a relationship between scope and binding: a NF-attribute must be bound to a component finer than its scope. So, it is not possible for a NF-attribute bound to a library to have a scope restricted to just a few modules of the library.
- They may have *multiple definitions*. So, different projects may choose the more appropriate definition of NF-attributes.

The first two characteristics appear when the NF-attribute is defined inside a *NF-attribute module*; a single module may define more than one NF-attribute. In case of multiple definitions, each of them will appear at a different module.

Non-functional attributes may involve one or more *measurement units*, which model data sizes.

3.2. Non-functional behaviour

Once a component specification has been built, implementations for it may be written. Each implementation V for a given software component D should state its NF-behaviour with respect to the basic NF-attributes that are in use in D; values of derived NF-attributes are automatically computed.

Obviously, a crucial question here is how can be checked that the stated NF-behaviour really corresponds to the functional implementation. Up to now, we have not done any significant work here, although we are trying to develop a framework to compute the value of some NF-attributes in an automatic manner [6].

3.3. Non-functional requirements

NF-requirements are the mean to state conditions on implementations of software components. Syntactically, they are usual boolean expressions enriched with some new constructs for non-functionality, which are mainly some useful quantification over the value of NF-attributes. Their purpose is to express relationships between properties and to represent the environment where implementations are to be inserted. They can appear both in non-functional specifications, to state properties that every implementation of the type must fulfil, or in non-functional implementations, to state constraints on the implementations of the imported components.

NF-requirements allow to refine the classical definition of implementation correctness, which focus on functional aspects. Given a specification SP and an implementation IMP, both with functional and non-functional parts, SP = (FSP, NFSP) and IMP = (FI, NFI), we say that IMP is correct w.r.t. SP if:

- FI is (functionally) correct w.r.t. FSP, for a given correctness definition.
- All the NF-requirements stated in NFSP are satisfied by NFI.

This second condition may be not computable if the NF-behaviour depends on the implementations of the imported components, as it happens often. In other words, a NF-requirement may not only hold or fail for a given implementation, but it also can be undefined. In this last case, we consider the implementation conditionally correct. A conditionally correct may later become a correct one, with appropriate implementation selection for the imported components.

4. Examples

We give here an example of treatment of two particular NF-requirements that appear in the meeting scheduler case study [7]: little interaction and good performance. We are going to develop in detail the first case, while we will just address partially to the second one, focusing on non-functional aspects. We use the constructs of NoFun to systematise non-functional descriptions.

To simplify some aspects of the development, we assume that the meeting initiator is always the same person.

4.1. Interaction

Meeting scheduler requirements include the sentence: "The amount of interaction among participants (...) should be kept as small as possible". We can view this as a NF-requirement concerning the usability of the system, and so we decide to deal with him in the non-functional side of system architecture, by defining a NF-attribute called *interaction* and then stating the valid NoFun constraint:

```
min(interaction)
```

as NF-requirement of the system.

4.1.1 Defining the NF-requirement. As suggested in [7], we consider interaction as the disjoint union of the number of delivered messages and their length (amount of negotiation could be expressed in terms of these two factors), trying then to measure the effort for obtaining the information for the meeting[2]:

```
union interaction = (delivered_messages,
                     information_volume)
```

So, minimising interaction means minimising both of them. This is to say, the following implicit equality follows from the *interaction* declaration:

```
min(interaction) = min(delivered_messages) ∧
                   min(information_volume)
```

While the first factor is clearly measurable with an integer, the way of dealing with the second is not so obvious. We decide to introduce an enumeration type trying to classify the most common information volume flow types:

```
integer delivered_messages
enumeration ordered information_volume =
                   (yes_no, form, ascii, multimedia, ...)
```

The key word "ordered" means that values of the domain can be compared with a total order operator, considering values as ordered as they appear in the definition. This gives sense to the expression *min(information_volume)*.

However, *min(interaction)* is not really a useful criterion for measuring. How do we know that a system meets this goal? In fact, the NoFun "min" operator makes sense when selecting between existing implementations, as we will see in 4.2. When just one implementation is expected, it is necessary to translate this concept into concrete values, and this is what we are doing next.

Let's start with *delivered_messages*. Concerning the functionality of the system, we can distinguish three kinds of interaction: initial information collection, resolution of conflicts and schedule broadcasting. As the exact number of messages depend on both the number of potential attendees and the number of conflicts, we introduce two measurement units for representing them:

```
measurement unit nb_attendees, nb_conflicts
```

Then, we obtain[3]:

- Initial information collection: a broadcasting message from the initiator to the other attendees and their response gives *nb_attendees* + 1 messages.
- Resolution of conflicts: the initiator sends a message to every attendee provoking a conflict, which must be answered. This yields to an amount of *nb_conflicts**2 messages.
- Schedule broadcasting: just a final message once all the conflicts have been solved.

[2] We are not considering explicitly the cost of reading messages. Partly, this cost depends on message length. However, it could be explicitly considered by introducing a third factor, or by redefining *delivered_messages* by *delivered_and_received_messages*.

[3] A particular point has not been considered here: data range extension, which causes iteration of the three steps presented above. A way of dealing with data range extension would be introducing a third measurement unit and then multiplying with the amount obtained below.

So, the optimal amount of messages equals to *nb_attendees* + *nb_conflics**2 + 2. However, requiring exactly this number of messages would certainly be too restrictive, so we allow an interaction policy with a few more messages:

```
requirements delivered_messages <
                    nb_attendees*2 + nb_conflicts*3
```

The important fact is that coefficients of measurement units remain the same, because they are the ones that rule the magnitude. With this requirement, we allow for instance the initiator delivering periodical messages to attendees informing of current status while negotiation is going on (just a few more messages during scheduling). On the other hand, some schemes do not pass this requirement and so they are forbidden; we may cite direct conflict resolution between conflicting attendees, which would increase the coefficient of *nb_conflicts*.

Concerning *information_volume*, we just require messages being ascii, forms or yes/no messages. Making use of the ordering relationship, this can be expressed as:

```
requirements information_volume <= ascii
```

4.1.2 NF-behaviour in a particular software architecture. Our software architecture for the meeting scheduler system (fig. 2) assigns a component for every attendee and for the initiator. In the components, we identify some ports that are bound with a connector.

```
component INITIATOR
    ports
        out init_meeting, notify_conflict
        out broadcast_meeting
        in receive_preferences, get_new_preferences
end INITIATOR

component ATTENDEE
    ports
        in receive_proposal, receive_meeting
        in conflict_notification
        out give_preferences, solve_conflict
end ATTENDEE

connector INITIATOR_ATTENDEE
    connects INITIATOR with ATTENDEE
    binds init_meeting with receive_proposal
          notify_conflict with conflict_notification
          broadcast_meeting with receive_meeting
          receive_preferences with give_preferences
          get_new_preferences with solve_conflict
end INITIATOR_ATTENDEE
```

Fig. 2: Components and connectors of the first layer of the meeting scheduler system architecture.

Given these ports with their obvious meaning, and given also the analysis of message flow done in the last section, the NF-behaviour of the system with respect to the number of delivered messages can be defined in the following way:

```
MEETING_SCHEDULER.delivered_messages =
    INITIATOR.init_meeting.delivered_messages
                    +
    ATTENDEE.give_preferences.delivered_messages
                *nb_attendees
                    +
    INITIATOR.notify_conflict.delivered_messages
                *nb_conflicts
                    +
    ATTENDEE.solve_conflict.delivered_messages
                *nb_conflicts
                    +
    INITIATOR.broadcast_meeting.deliverd_messages
```

(Note qualification of NF-attributes to distinguish which software unit are they referring to.)

Concerning message volume, and taking profit of the fact of the NF-attribute being ordered, we define it as the maximum of the values in those operations involving more than one message:

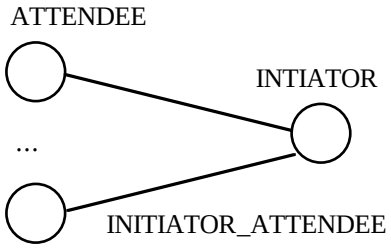
```
MEETING_SCHEDULER.information_volume =
        max(information_volume,
                {ATTENDEE.give_preferences,
                 INITIATOR.notify_conflict,
                 ATTENDEE.solve_conflict})
```

The last step has to be with giving values to the NF-attributes at the port level. This should appear in the NF-behaviour modules bound to the *ATTENDEE* and *INITIATOR* components. The values should be explicitly written by humans, until a way of computing them automatically is provided.

Fig. 3 presents the whole first layer of the meeting scheduler system architecture.

4.2. Performance

One of the non-functional goals of the system is performance. Although this goal must be carefully handled, especially with colliding with other goals, some particular requirements can be easily identified. Consider, for instance, a partial view of the inner architecture for the *INITIATOR* component, formed by a control module and a data repository for storing user names. The control module imports the repository and so there exists a

<table>
<tr>
<td>Functionality of the
system as described in [7]</td>
<td>ATTRIBUTES
UNION interaction = (delivered_messages, information_volume)
INTEGER delivered_messages
ENUMERATION information_volume = (yes_no, …)
MEASUREMENT UNITS nb_attendees, nb_conflicts
REQUIREMENTS
delivered_messages < nb_attendees*2 + nb_conflicts*3
information_volume <= ascii</td>
</tr>
<tr>
<td> 

described as in fig. 2</td>
<td>MEETING_SCHEDULER.delivered_messages =
INITIATOR.init_meeting.delivered_messages +
ATTENDEE.give_preferences.delivered_messages*nb_attendees +
INITIATOR.notify_conflict.delivered_messages*nb_conflicts +
ATTENDEE.solve_conflict.delivered_messages*nb_conflicts +
INITIATOR.broadcast_meeting.delivered_messages

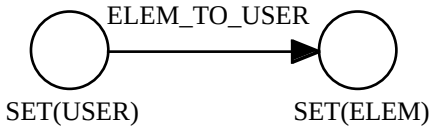
MEETING_SCHEDULER.information_volume =
max(information_volume, {ATTENDEE.give_preferences,
INITIATOR.notify_conflict,
ATTENDEE.solve_conflict})</td>
</tr>
</table>

Fig. 3: First layer of the meeting scheduler system architecture (left column: functional; right column: non-functional; upper row: specification: lower row: implementation).

data repository for storing user names. The control module imports the repository and so there exists a connector between both of them. If we define the repository as an instance of a generic *SET* component, this connector can be defined in terms of a parameter passing operation that bounds user characteristics to set elements.

If many implementations for sets exist using different data structures (hashing, AVL trees, lists, etc.), the main criteria for choosing one of them seems to be efficiency, perhaps together with reliability. To be more precise, it seems advisable to minimise the execution time for the *retrieval* operation, which surely will be the most frequently used. This requirement will be more important if the component is going to be adapted to other environments, perhaps larger ones, as suggested by the system requirements in [7]. As a secondary criterion, we ask for error recovering in implementations.

It is important to note that this NF-requirement must appear in the non-functional part of the connector that represents the import relationship, as shown in fig. 4. Note also that the implementation part of the connector is empty, because the parameter passing really takes place in the *SET(USER)* component using the constructs provided for whatever the programming language is, and also because we have not included NF-attributes specific for the connector to assign values to.



association set element --> user	**requirements** min(time(retrieval)) and reliability < low; error_recovery
Ø	Ø

ELEM_TO_USER

Fig. 4: Defining a set of users from generic sets.

5. Conclusions

We have presented a proposal for putting non-functional information of software systems into software architectures. Non-functionality is described by means of a notation called NoFun, which allows to introduce non-functional attributes of software, to give values to these attributes in component and connectors, and to formulate non-functional requirements in terms of these attributes. We have tried to illustrate the feasibility of our approach

by addressing two particular aspects of the meeting scheduler case study.

We consider that the salient features of our approach are:

- NoFun is a step forward to deal with non-functionality in a precise way, different from the usual case (natural language). There is a lot of work done in studying NF-attributes, defining metrics, and so on, but we think there is a lack of notations to express the concepts arising in the field. A notation such as NoFun provides a common framework in which people can state, analyse and compare their proposals about non-functionality.
- Concerning the power of the language, we would like to remark that it presents many features which are necessary to model non-functionality in a proper way: 1) NF-attributes may be defined in more than one way; 2) they can be bound to different software units; 3) they can have different scopes; 4) NF-requirements may be ordered with respect to their relative importance.
- We have stated our approach without compromising with particular software architecture styles or notation. So, we think that our approach can be adapted to a variety of particular existing proposals.

There are many approaches for defining a language to state non-functionality, but as far as we know they are limited in scope. An interesting approach appears in [23], which provides a framework to evaluate the design of software systems, the measurement criterion being the adequacy of implementations with respect to some non-functional requirements stated over a set of attributes. The requirements are stated as an array of weights over the properties and every attribute has a weight too; then, the evaluation of implementations results in a number and comparison is possible. However, the notation proposed in this work is not as general as that presented here; also, the proposal is not integrated into the software itself losing some of the advantages we have mentioned in the introduction.

On the other hand, [5, 16, 19, 21] provide some language constructs to state program efficiency. [5] aims at coding generation from some high-level language constructs manipulating a *relation* data type; in the general case, there are many ways to generate this code and so information about efficiency is used to select the optimal translation. [21] focuses on program transformation: algorithms are refined using a library of components with pre-post functional specifications; when there are many components whose pre-post specification allows its inclusion in the algorithm being refined, efficiency is used to break the tie. The proposal in [16] keeps similar ideas but in the real-time framework.

Concerning [19], it is perhaps the proposal closest to ours due to its definition in the component programming framework and also to the existence of special modules collecting some kind of non-functional information (constraints on efficiency in this case), although the work focuses on software reusability and verification, which are two fields we have not yet addressed. Efficiency in [19] is difficult to handle because it is "tight" efficiency (an exact measure of efficiency) and this often requires the definition of auxiliary models to express the time consumed by component operations.

Also, it is worth mentioning the existence of the so-called process-oriented approaches for dealing with non-functionality. These approaches use non-functional information to guide the development of software systems. The most widespread one was proposed by Mylopoulos *et al.* [15] and it was developed in the information systems area. Also, some work has been done on knowledge-based systems, notably in the MIKE system [24], whose main ideas are close to those of Mylopoulos *et al*. In both cases, they also propose a language more oriented to keep track of the development process, defining some constructs to refine non-functional goals, to state conflicting NF-requirements, and by the like. In some sense, we can say that their language is more abstract than ours, and as stated in [15], we could see both notations as complementary, both contributing to a comprehensive framework for dealing with non-functional specifications and non-functional requirements.

Acknowledgments

We would like to thank the anonymous referees for their valuable comments, as well as A. Finkelstein for his suggestions.

References

[1] G.D. Abowd, R. Allen, D. Garlan. "Formalizing Style to Understand Descriptions of Software Architecture". *ACM Transactions on Software Engineering and Methodology*, 4(4), pp. 319-364, October 1995.

[2] G. Berry, G. Boudol. "The Chemical Abstract Machine". *Theoretical Computer Science*, 96, pp. 216-248, 1992.

[3] B. Boehm, H. In. "Identifying Quality-Requirements Conflicts". IEEE Software March 1996, pp. 25-35.

[4] G. Caldiera, V.R. Basili. "Identifying and Qualifying Reusable Software Components". IEEE Computer, 24(2), 1991.

[5] D. Cohen, N. Goldman, K. Narayanaswamy. "Adding Performance Information to ADT Interfaces". In *Proceedings of the Interface Definition Languages Workshop*, ACM SIGPLAN Notices 29(8), 1994.

[6] X. Franch, P. Botella, X. Burgués, J.M. Ribó. "ComProLab: A Component Programming Laboratory". In *Proceedings of 9th Software Engineering and Knowledge Engineering Conference (SEKE)*, Madrid (Spain), 1997.

[7] M.S. Feather, S. Fickas, A. Finkelstein, A. van Lansweerde. "Requirements and Specification Exemplars". *Automated Software Engineering*, 1997.

[8] X. Franch. "Systematic Formulation of Non-Functional Characteristics of Software". In *Proceedings of 3rd International Conference on Requirements Engineering* (ICRE), Colorado Springs (USA), 1998.

[9] J.V. Guttag, J.J. Horning. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science, Springer-Verlag, 1993.

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[11] D. Garlan, M. Shaw. "An Introduction to Software Architecture". In *Advances in Software Engineering and Knowledge Engineering*, pp. 1-39, Eds. World Scientific, 1993.

[12] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[13] IEEE Computer Society. *IEEE Standard for a Software Quality Metrics Methodology*. IEEE Std. 1061-1992, Institute of Electrical and Electronical Engineers, New York, 1992.

[14] International Standards Organization. *Software Product Evaluation - Quality Characteristics and Guidelines for their Use*. ISO/IEC Standard ISO-9126, 1991.

[15] J. Mylopoulos, L. Chung, B.A. Nixon. "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach". IEEE Transactions on Software Engineering, 18(6), 1992.

[16] R.H. Pierce *et al*. "Capturing and verifying performance requirements for hard real-time systems". In *Proceedings International Conference on Software Reliable Technologies*, London (England), LNCS 1251, Springer-Verlag, 1997.

[17] G.-C. Roman. "A Taxonomy of Current Issues in Requirements Engineering". IEEE Computer, 18(4), 1985.

[18] M. Shaw. "Abstraction Techniques in Modern Programming Languages". IEEE Software, 1(10), 1984.

[19] M. Sitaraman. "On Tight Performance Specification of Object-Oriented Components". In *Proceedings 3rd International Conference on Software Reuse (ICSR)*, IEEE Computer Society Press, 1994.

[20] J.M. Spivey. *The Z Notation*. Prentice-Hall, 1993.

[21] P.C-Y. Sheu, S. Yoo. "A Knowledge-Based Program Transformation System". In *Proceedings 6th CAiSE*, Utrecht (Holland), LNCS 811, 1994.

[22] J.M. Wing. "A Specifier's Introduction to Formal Methods". IEEE Computer 23(9), 1990.

[23] S. Cárdenas, M.V. Zelkowitz. "Evaluation Criteria for Functional Specifications". In *Proceedings of 12th ICSE*, Nice (France), 1990.

[24] D. Landes, R. Studer. "The Treatment of Non-Functional Requirements in MIKE". In *Proceedings of 5th ESEC*, Barcelona (Catalunya, Spain), LNCS 989, Springer-Verlag, 1995.